Padrões de projeto

Alunos: Gabriel Brandão, Leandro Alves, Caique Silva e Bruno Bispo
Professora: Larissa Rocha

Sumário

- O que são Padrões de Projeto?
- Padrão de Projeto Builder
- Padrão de Projeto Composite
- Padrão de Projeto Strategy
- Padrão de Projeto Mediator

O que são Padrões de Projeto?

- O que é?

São soluções para problemas de projetos de softwares. Cada padrão é como uma planta pré-projetada que se pode alterar na resolução de problemas em um software.

Tipos:

- Criacionais: Padrões que possibilitam mais de uma opção na criação de objetos, aumentando flexibilidade e reutilização do código.
- Estruturais: Padrões para montagem de classes e objetos sem perder a flexibilidade e eficiência
- Comportamentais: Padrões que ajudam no trabalho com os algoritmos e delegação de funções aos objetos.

Padrão de Projeto Builder

O QUE É?

Padrão de Projeto Builder

Principais motivos para usar:

1- Evitar construtores confusos com muitos parâmetros

2 - Facilitar a leitura e manutenção

3 - Validações na construção

4 - Facilitar a criação de objetos com variações

Padrão de Projeto Builder

Prós:

Clareza e legibilidade: Evita construtores com muitos parâmetros, facilitando a leitura e manutenção.

Imutabilidade: Facilita a criação de objetos imutáveis com muitos atributos.

Flexibilidade: Permite criar diferentes versões do mesmo objeto (com ou sem certos atributos).

Encapsulamento da lógica de construção: A lógica de validação ou transformação de dados pode ficar centralizada no Builder.

Contra:

Mais código: Cria uma nova classe (ou classe interna) para o Builder, o que aumenta a quantidade de código.

Pode ser exagero para objetos simples: Se o objeto só tem 2 ou 3 campos, o Builder pode ser desnecessário.

Duplicação de campos: Os campos precisam ser definidos tanto na classe principal quanto no Builder.

Padrão de Projeto Builder

EXEMPLO:

Crie uma classe Pedido para um sistema de e-commerce, com os seguintes campos:

`cliente (obrigatório)`

`enderecoEntrega (obrigatório)`

`itens`

`freteGratuito`

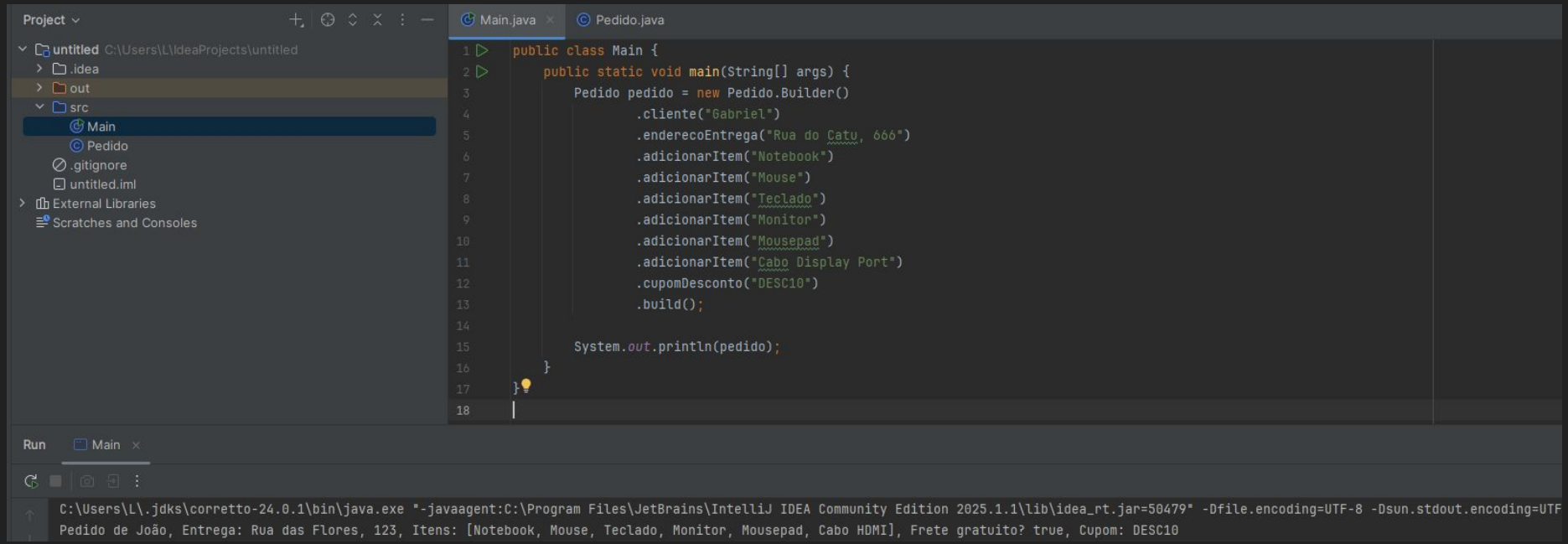
`cupomDesconto (opcional)`

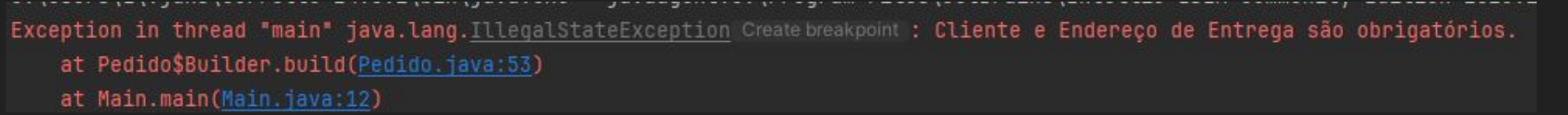
Regras:

Se o total de itens for mais que 5, freteGratuito deve ser forçado como true, mesmo que o usuário não defina.

O método build() deve lançar IllegalStateException se cliente ou enderecoEntrega forem nulos.

```java
import java.util.ArrayList;
import java.util.List;

public class Pedido {  2 usages
    private String cliente;  2 usages
    private String enderecoEntrega;  2 usages
    private List<String> itens;  2 usages
    private boolean freteGratuito;  2 usages
    private String cupomDesconto;  3 usages

    private Pedido(Builder builder) {  1 usage
        this.cliente = builder.cliente;
        this.enderecoEntrega = builder.enderecoEntrega;
        this.itens = builder.itens;
        this.freteGratuito = builder.itens.size() > 5 ? true : builder.freteGratuito;
        this.cupomDesconto = builder.cupomDesconto;
    }

    public static class Builder {  6 usages
        private String cliente;  3 usages
        private String enderecoEntrega;  3 usages
        private List<String> itens = new ArrayList<>();  3 usages
        private boolean freteGratuito = false;  2 usages
        private String cupomDesconto;  2 usages

        public Builder cliente(String cliente) {  no usages
            this.cliente = cliente;
            return this;
        }

        public Builder enderecoEntrega(String endereco) {  no usages
            this.enderecoEntrega = endereco;
            return this;
        }

        public Builder adicionarItem(String item) {  no usages
            this.itens.add(item);
            return this;
        }

        public Builder freteGratuito(boolean freteGratuito) {  no usages
            this.freteGratuito = freteGratuito;
            return this;
        }

        public Builder cupomDesconto(String cupom) {  no usages
            this.cupomDesconto = cupom;
            return this;
        }

        public Pedido build() {  no usages
            if (cliente == null || enderecoEntrega == null) {
                throw new IllegalStateException("Cliente e Endereço de Entrega são obrigatórios.");
            }
            return new Pedido( builder: this);
        }
    }

    @Override
    public String toString() {
        return "Pedido de " + cliente + ", Entrega: " + enderecoEntrega +
                ", Itens: " + itens + ", Frete gratuito? " + freteGratuito +
                ", Cupom: " + (cupomDesconto != null ? cupomDesconto : "Nenhum");
    }
}
```


Project
untitled C:\Users\L\IdeaProjects\untitled
.idea
out
src
Main
Pedido
.gitignore
untitled.iml
External Libraries
Scratches and Consoles
Main.java
Pedido.java
public class Main {
public static void main(String[] args) {
Pedido pedido = new Pedido.Builder()
.cliente("Gabriel")
.enderecoEntrega("Rua do Catu, 666")
.adicionarItem("Notebook")
.adicionarItem("Mouse")
.adicionarItem("Teclado")
.adicionarItem("Monitor")
.adicionarItem("Mousepad")
.adicionarItem("Cabo Display Port")
.cupomDesconto("DESC10")
.build();
System.out.println(pedido);
}
}
Run
Main
C:\Users\L\.jdks\corretto-24.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2025.1.1\lib\idea_rt.jar=50479" -Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF
Pedido de João, Entrega: Rua das Flores, 123, Itens: [Notebook, Mouse, Teclado, Monitor, Mousepad, Cabo HDMI], Frete gratuito? true, Cupom: DESC10


Exception in thread "main" java.lang.IllegalStateException Create breakpoint : Cliente e Endereço de Entrega são obrigatórios.
at Pedido$Builder.build(Pedido.java:53)
at Main.main(Main.java:12)

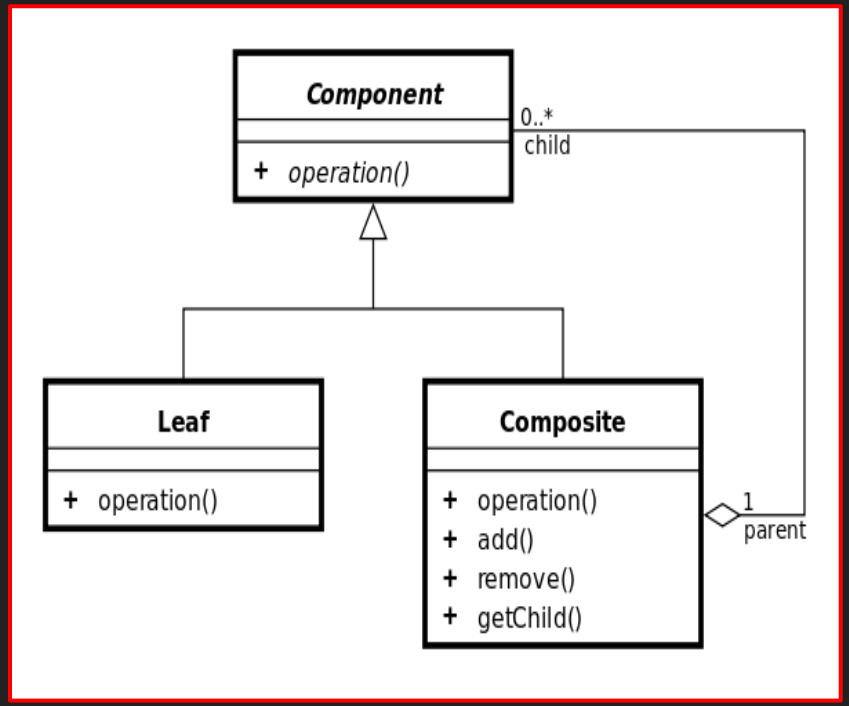
Padrão de Projeto Composite

O **Composite** é um padrão de projeto **estrutural** que permite que você componha objetos em *estruturas de árvores* e então trabalhe com essas estruturas como se elas fossem objetos individuais.

Hierarquia de objetos: Permite construir árvores de objetos onde objetos individuais e compostos são tratados da mesma forma.

Unificação das operações: Os métodos são aplicados tanto a objetos únicos quanto a coleções de objetos.

Flexibilidade e extensibilidade: Facilita a adição de novos tipos de componentes sem alterar código existente.

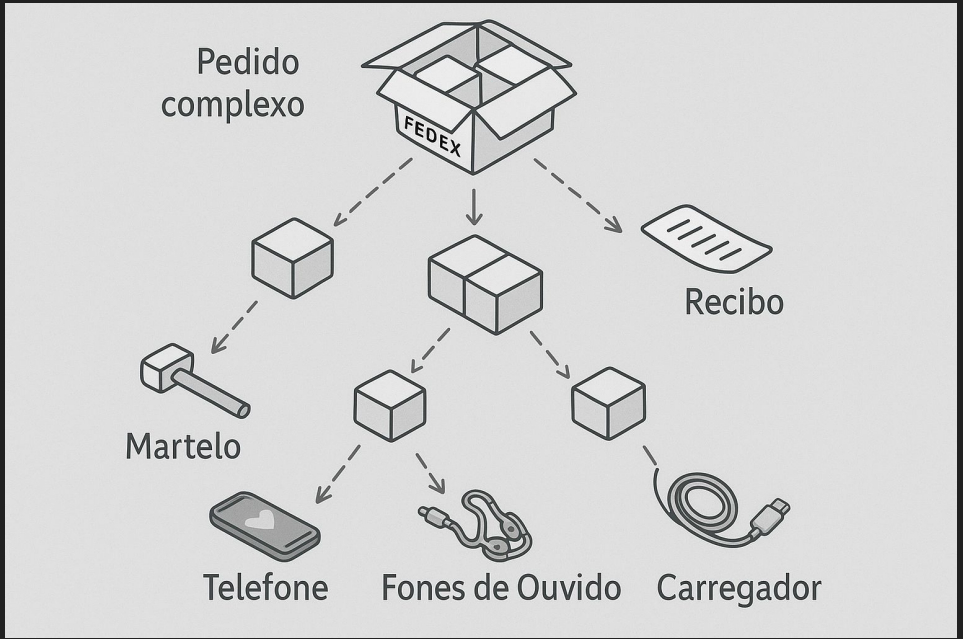


O Problema

Como calcular o preço total de um pedido com produtos e caixas aninhadas?

Uma solução direta seria desempacotar tudo e calcular.

Mas isso exige conhecer classes, níveis de aninhamento e outros detalhes.



A Solução

Interface Comum

Trabalhe com produtos e caixas através de uma interface comum.

Cálculo Recursivo

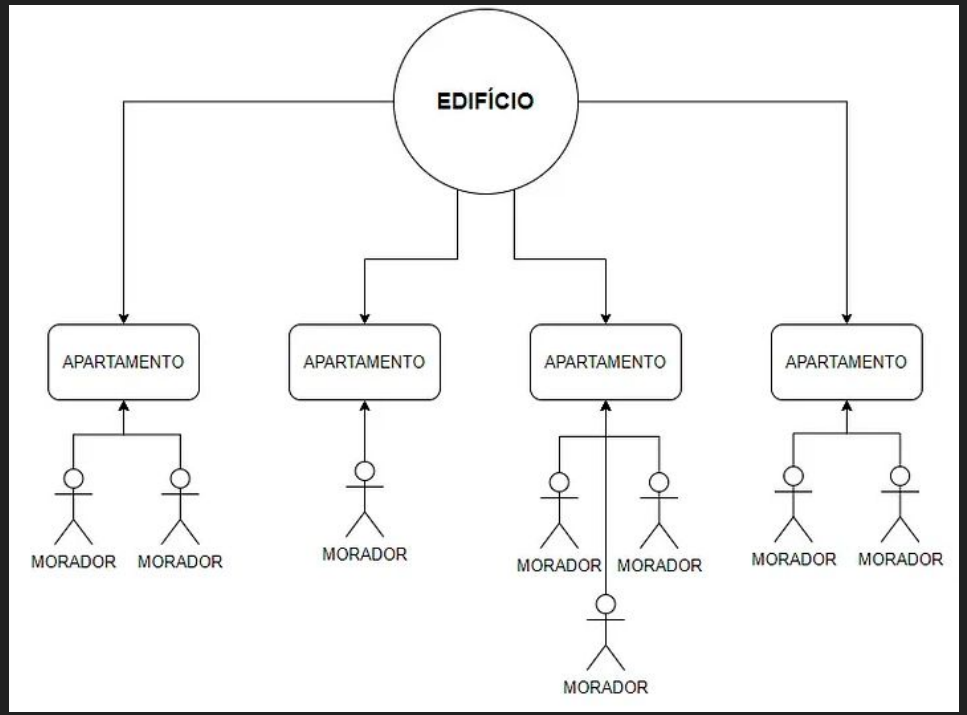
Produtos retornam seu preço. Caixas somam preços de seus itens.

Estrutura em Árvore

O comportamento é executado recursivamente em todos os componentes.

Exemplo

Na estrutura de um edifício representado no diagrama, o Edifício seria um **Composite** pois é composto de um grupo de apartamentos, já Apartamento seria um **Composite** também pois compõe de um ou mais moradores e o Morador seria um **Leaf** pois é uma unidade isolada

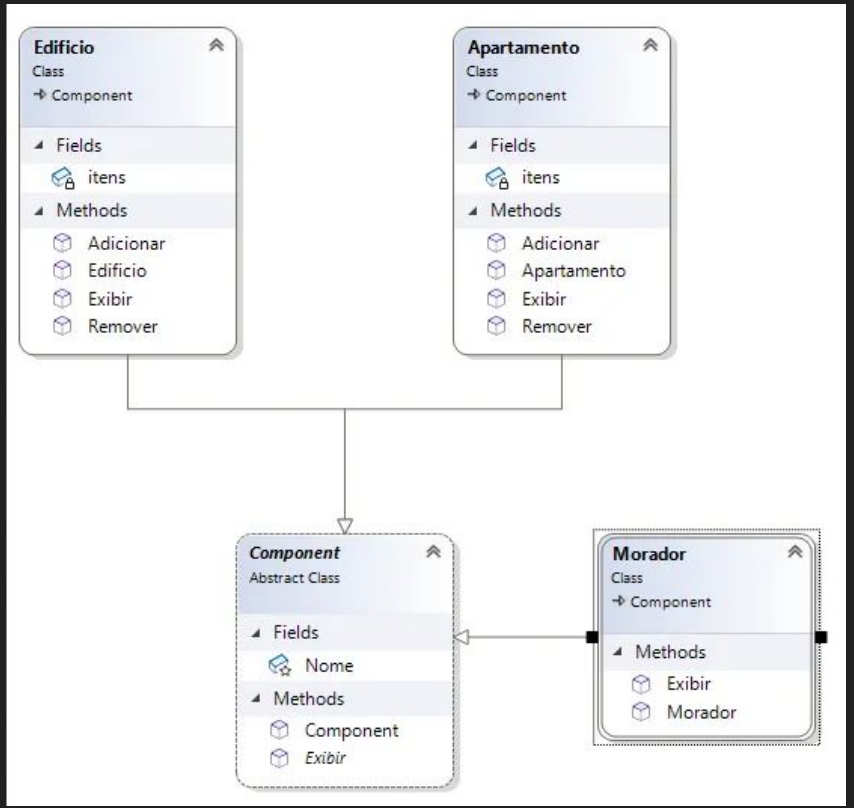


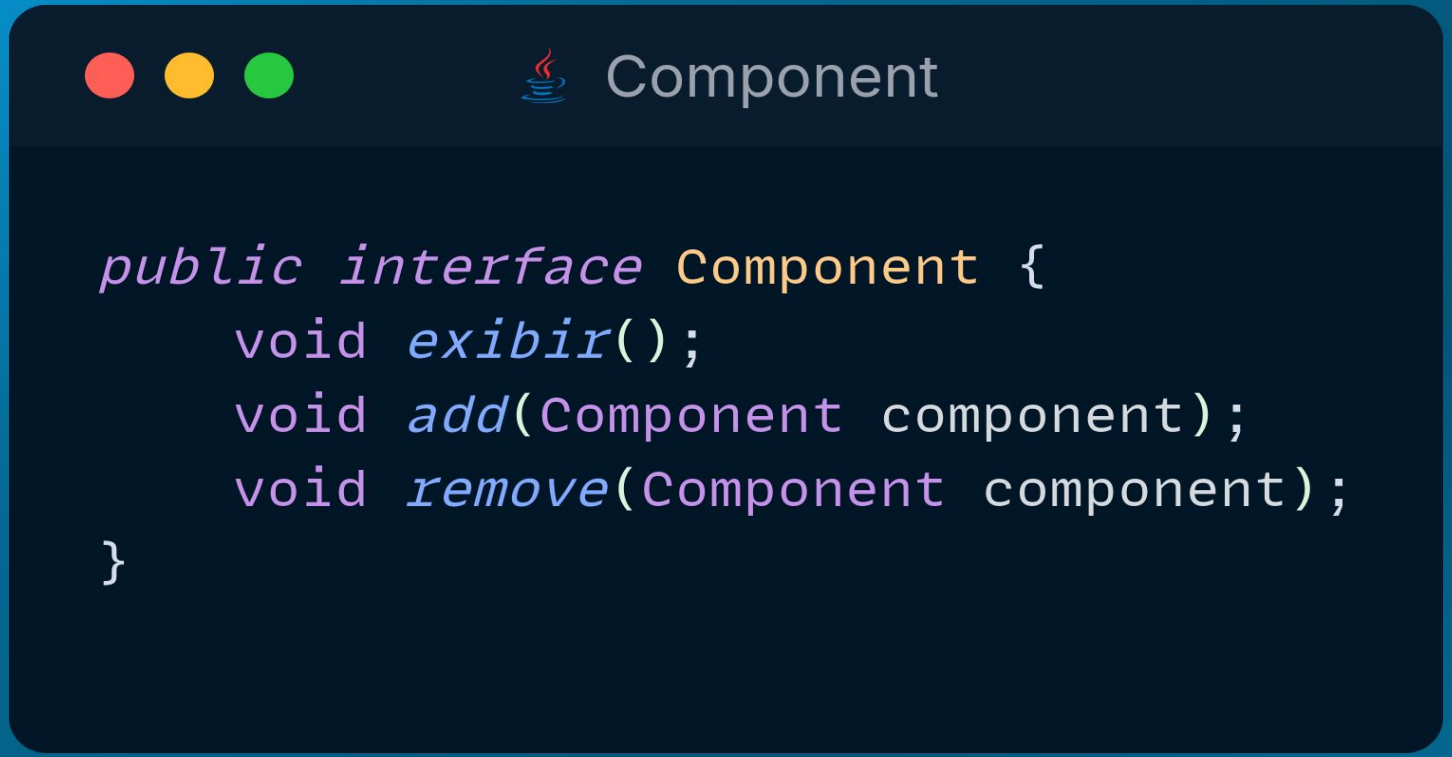
Exemplo

- **Component** — interface que descreve operações em comum aos elementos simples (**Leaf**) e estruturados/compostos (**Composite**).

- **Composite** — objeto composto por outros objetos, ou seja, algo composto, ramificado ou estruturado.

- **Leaf** — na tradução literal seria a folha, no caso a folha da árvore, também a extremidade mais simples e não possui elementos abaixo.




Component
public interface Component {
void exibir();
void add(Component component);
void remove(Component component);
}

Morador

```java
public class Morador implements Component {
    private final String nome;

    public Morador(String nome) {
        this.nome = nome;
    }

    @Override
    public void exibir() {
        System.out.println("Morador: " + nome);
    }

    @Override
    public void add(Component component) {
        throw new UnsupportedOperationException("Não é possível adicionar a um morador");
    }

    @Override
    public void remove(Component component) {
        throw new UnsupportedOperationException("Não é possível remover de um morador");
    }
}
```

Edificio

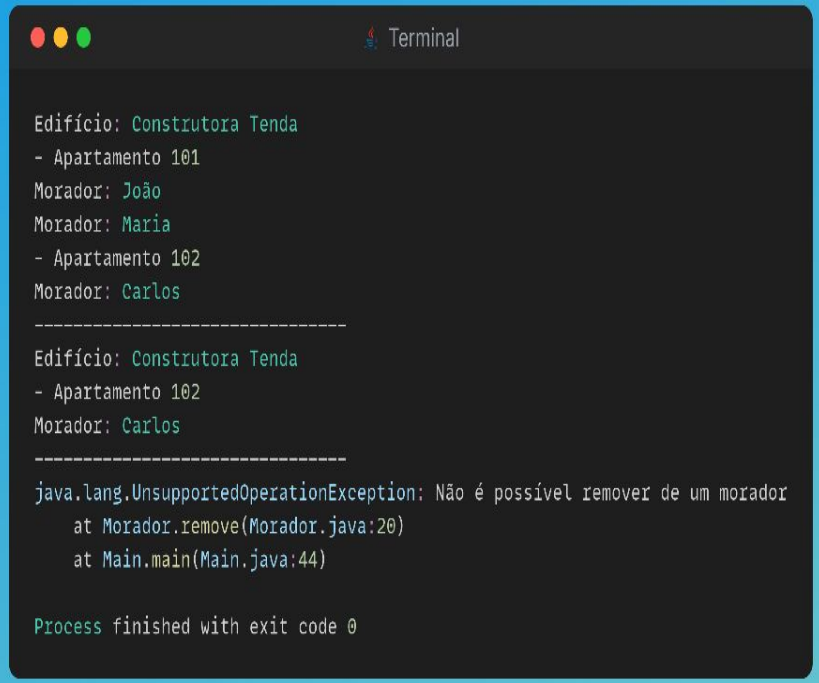
```java
import java.util.ArrayList;
import java.util.List;

public class Edificio implements Component {
    private final String nome;

    private final List<Component> apartamentos = new ArrayList<>();

    public Edificio(String nome) {
        this.nome = nome;
    }

    @Override
    public void exibir() {
        System.out.println("Edifício: " + nome);
        for (Component component : apartamentos) {
            component.exibir();
        }
    }

    @Override
    public void add(Component component) {
        apartamentos.add(component);
    }

    @Override
    public void remove(Component component) {
        apartamentos.remove(component);
    }
}
```

Apartamento

```java
import java.util.ArrayList;
import java.util.List;

public class Apartamento implements Component {
    private final String numero;
    private final List<Component> moradores = new ArrayList<>();

    public Apartamento(String numero) {
        this.numero = numero;
    }

    @Override
    public void exibir() {
        System.out.println("- Apartamento " + numero);
        for (Component component : moradores) {
            component.exibir();
        }
    }

    @Override
    public void add(Component component) {
        moradores.add(component);
    }

    @Override
    public void remove(Component component) {
        moradores.remove(component);
    }
}
```

Main

```java
public class Main {
    public static void main(String[] args) {
        // Cria um edifício com o nome "Construtora Tenda"
        Edificio e = new Edificio("Construtora Tenda");

        // Cria um apartamento com número 101
        Apartamento ap101 = new Apartamento("101");

        // Cria dois moradores: João e Maria
        Morador joao = new Morador("João");
        Morador maria = new Morador("Maria");

        // Adiciona João e Maria ao apartamento 101
        ap101.add(joao);
        ap101.add(maria);

        // Adiciona o apartamento 101 ao edifício
        e.add(ap101);

        // Cria outro apartamento (102) e adiciona um morador (Carlos)
        Apartamento ap102 = new Apartamento("102");
        ap102.add(new Morador("Carlos"));

        // Adiciona o apartamento 102 ao edifício
        e.add(ap102);

        // Exibe toda a estrutura do edifício: apartamentos e seus moradores
        e.exibir();

        System.out.println("---------------------------------");

        // Remove o apartamento 101 do edifício
        e.remove(ap101);

        // Exibe novamente a estrutura do edifício, agora sem o apartamento 101
        e.exibir();

        System.out.println("---------------------------------");

        // Tenta remover Maria a partir de João
        // Isso deve lançar uma exceção, pois "Morador" é uma folha (leaf)
        // e não pode conter ou remover outros componentes
        try {
            joao.remove(maria);
        } catch (UnsupportedOperationException exception) {
            // Captura e imprime o erro gerado pela tentativa de remover de um leaf
            exception.printStackTrace();
        }
    }
}
```



Prós e Contras

Vantagens

- Trabalho conveniente com estruturas de árvore complexas
- Uso eficiente de polimorfismo e recursão
- Respeita o princípio aberto/fechado

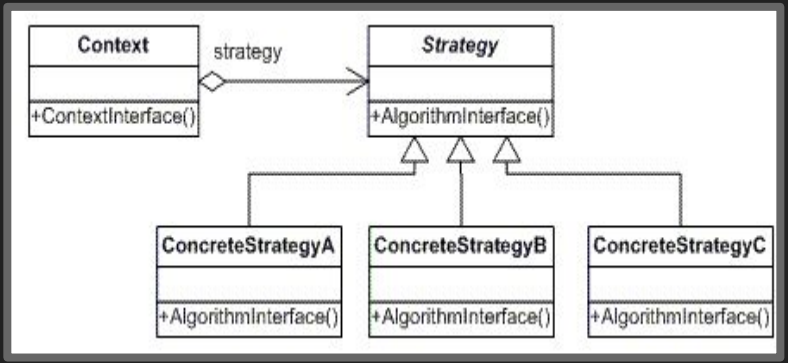
Desvantagens

- Pode ser difícil criar interface comum para classes muito diferentes
- Risco de generalizar demais a interface componente
- Possível complexidade na implementação

Padrão de Projeto Strategy

- O que é?

Padrão de projeto comportamental, que permite que se defina uma família de algoritmos, as separam em classes e faça os objetos intercambiáveis.



- Quando usar?

Solucionar problemas de várias classes que fazem a mesma ação, diferenciando apenas em sua aplicação. A solução é criar duas classes, uma contexto e outra estratégia (Strategy), que serão responsáveis respectivamente por, criar a ação que a estratégia fará e especificar o que a sua ação faz dentro do que é pedido.

Padrão de Projeto Strategy

Problema:

Adição de uma nova classe de mesmo intuito a cada nova atualização. Fazendo um código muito inchado e a cada vez mais difícil de adicionar novas funcionalidade.

Solução:

Um algoritmo que generaliza a ação e cria diferentes estratégias, para a cada nova classe, cada classe dessas irá se relacionar a um contexto, que trabalha com as estratégias de forma genérica.

Padrão de Projeto Strategy

Contras:

- Se você só tem um par de algoritmos e eles raramente mudam, não há motivo real para deixar o programa mais complicado com novas classes e interfaces que vêm junto com o padrão.
- Os Clientes devem estar cientes das diferenças entre as estratégias para serem capazes de selecionar a adequada.
- Muitas linguagens de programação modernas tem suporte do tipo funcional que permite que você implemente diferentes versões de um algoritmo dentro de um conjunto de funções anônimas. Então você poderia usar essas funções exatamente como se estivesse usando objetos estratégia, mas sem inchar seu código com classes e interfaces adicionais.

Prós:

- Você pode trocar algoritmos usados dentro de um objeto durante a execução.
- Você pode isolar os detalhes de implementação de um algoritmo do código que usa ele.
- Você pode substituir a herança por composição.
- *Princípio aberto/fechado*. Você pode introduzir novas estratégias sem mudar o contexto.

Padrão de Projeto Strategy

BattleStrategy.java

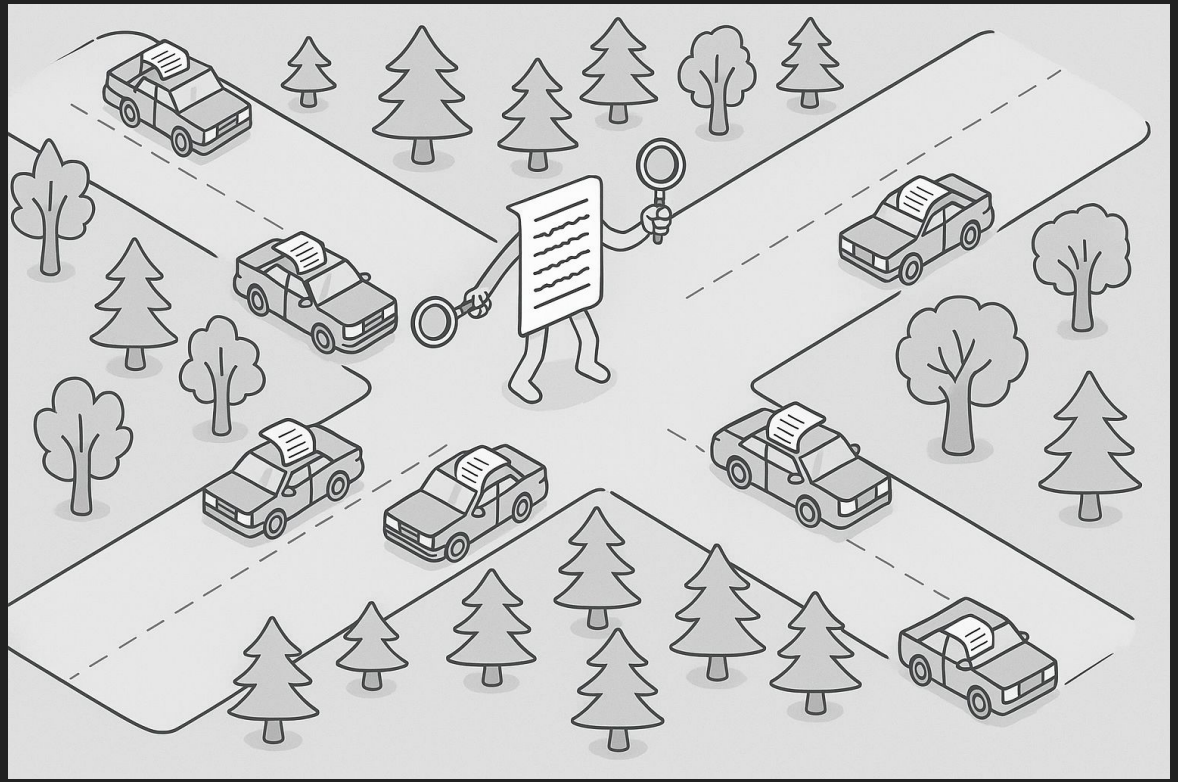
```java
package br.lawtrel.fantasia.battle;

import br.lawtrel.fantasia.entities.Character;

public interface BattleStrategy {  1 usage  3 implementations
    void executeBattle(Character actor, Character target);  3 usages  2 implementations
}

```

AttackStrategy.java

```java
package br.lawtrel.fantasia.battle;

import br.lawtrel.fantasia.entities.Character;

public class AttackStrategy implements BattleStrategy {  4 usages  2 inheritors
    @Override  3 usages  1 override
    public void executeBattle(Character actor, Character target){
        int damage = actor.getAttack() - target.getDefense();
        if(damage > 0){
            target.takeDamage(damage);
            System.out.println(actor.getName() + " atacou " + target.getName() + " causando " + damage + " de dano");
        }else{
            System.out.println("O ataque não causou nenhum dano LOL");
        }
    }
}
```

Padrão de Projeto Mediator

O Mediator é um padrão comportamental que reduz dependências caóticas entre objetos. Ele restringe comunicações diretas e força a colaboração através de um objeto mediador.



O Problema

Interações Complexas

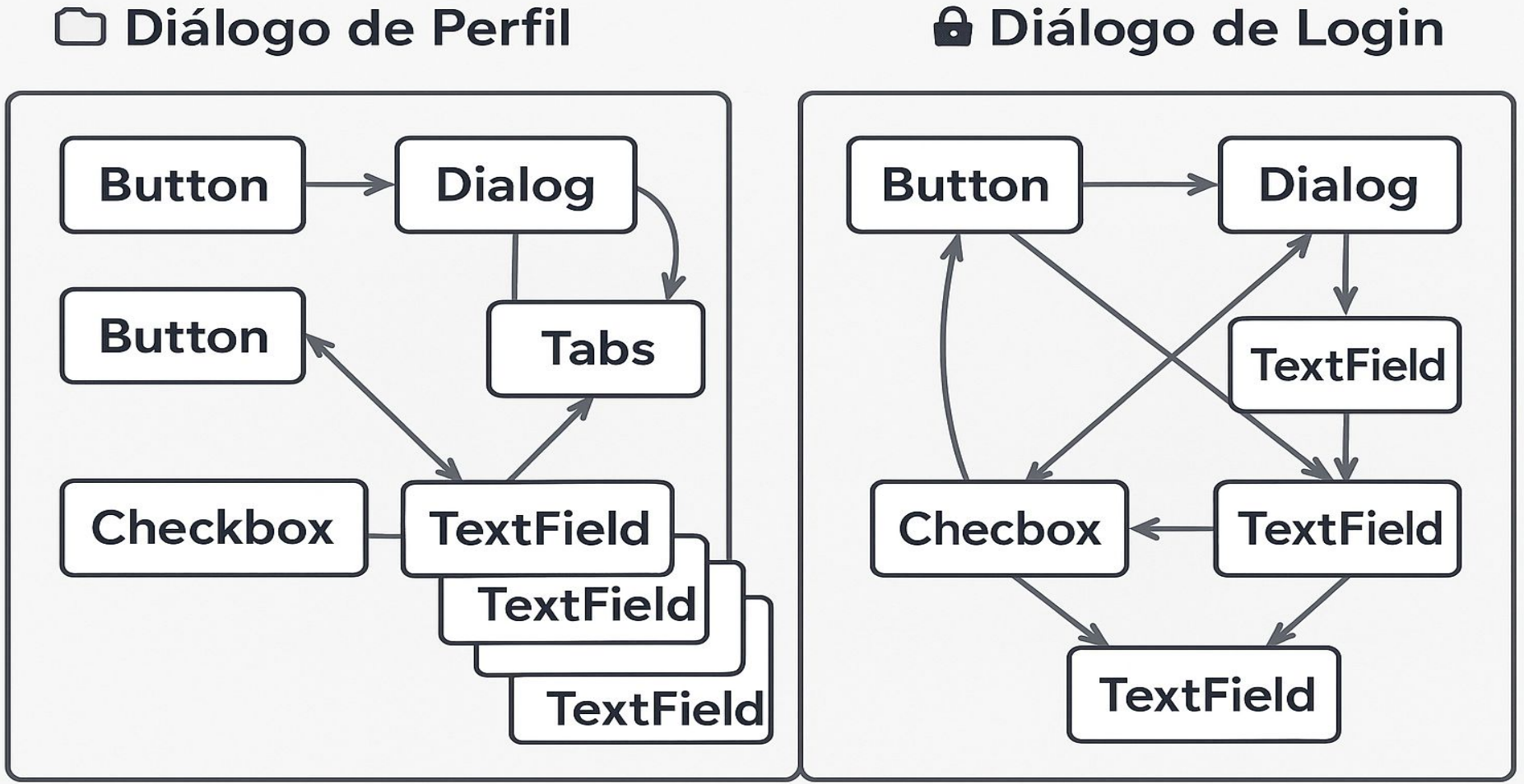
Elementos de interface podem ter múltiplas relações entre si, criando dependências caóticas.

Difícil Reutilização

Classes com lógica de interação direta tornam-se difíceis de reutilizar em outros contextos.

Acoplamento Forte

Mudanças em alguns elementos podem afetar muitos outros, complicando a manutenção.


Diálogo de Perfil
Button
Dialog
Button
Tabs
Checkbox
TextField
TextField
TextField
Diálogo de Login
Button
Dialog
TextField
Checbox
TextField
TextField

A Solução

1

Cessar Comunicação Direta

Componentes não se comunicam diretamente entre si.

2

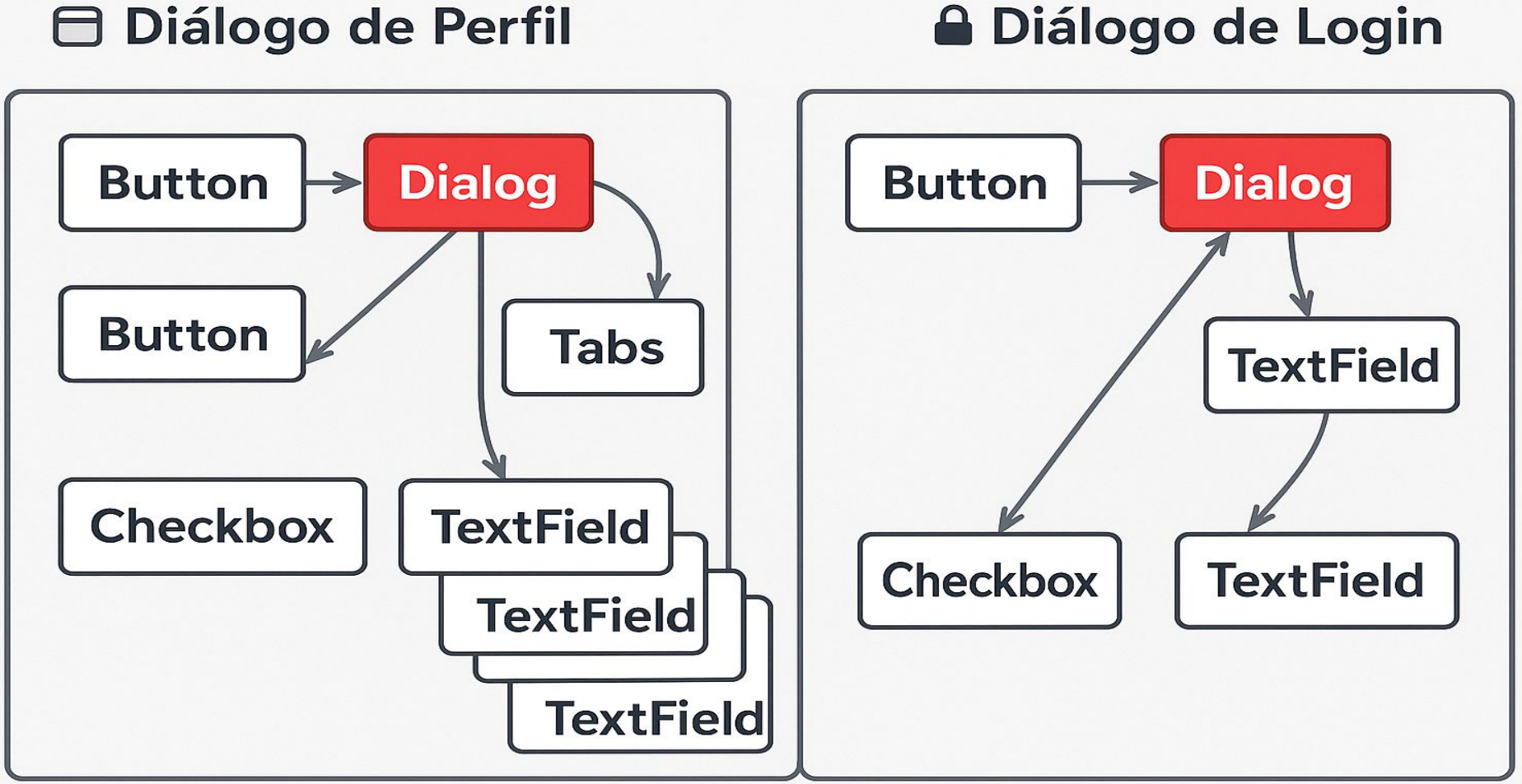
Introduzir Mediador

Um objeto mediador especial gerencia todas as interações.

3

Reduzir Dependências

Componentes dependem apenas da classe mediadora, não de múltiplos colegas.


Diálogo de Perfil
Button
Dialog
Button
Tabs
Checkbox
TextField
TextField
TextField
Diálogo de Login
Button
Dialog
TextField
Checkbox
TextField

Analogia do Mundo Real

Torre de Controle

Pilotos não se comunicam diretamente entre si para decidir quem aterrissa.

Segurança Aumentada

Este sistema evita caos e reduz significativamente o risco de acidentes.

Controlador Central

A torre de controle coordena todas as comunicações entre aeronaves.

Estrutura do Padrão

Componentes

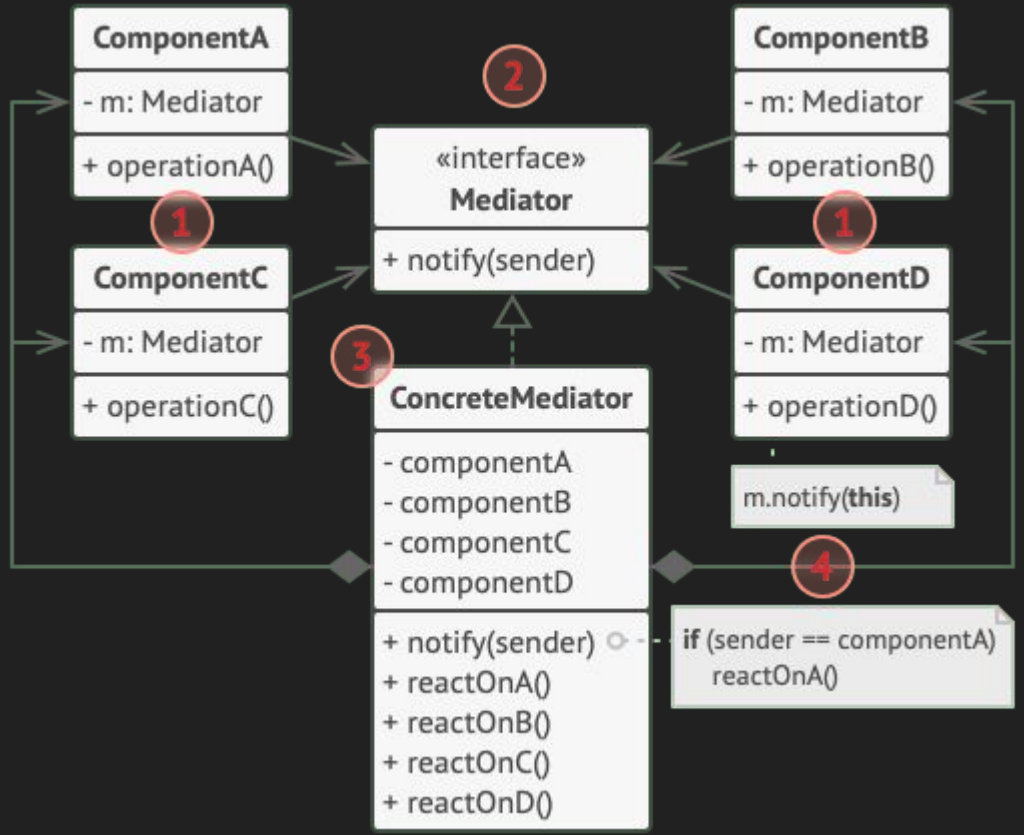
Classes com lógica de negócio que mantêm referência ao mediador através de uma interface.

Interface do Mediador

Declara métodos de comunicação, geralmente apenas um método de notificação.

Mediador Concreto

Encapsula relações entre componentes e mantém referências a todos eles.



Exemplo de implementação do mediator

```
1  // Interface do Mediador
2  interface Mediator {
3      void enviarMensagem(String mensagem, Participante remetente);
4  }
```

```java
// Implementação do Mediador
class MediadorConcreto implements Mediator {
    private List<Participante> participantes = new ArrayList<>();

    public void registrarParticipante(Participante participante) {
        participantes.add(participante);
        participante.definirMediador(this);
    }

    public void enviarMensagem(String mensagem, Participante remetente) {
        for (Participante participante : participantes) {
            if (participante != remetente) {
                participante.receberMensagem(mensagem);
            }
        }
    }
}
```

```java
// Classe abstrata Participante
abstract class Participante {
    protected Mediator mediador;

    public void definirMediador(Mediator mediador) {
        this.mediador = mediador;
    }

    public void enviarMensagem(String mensagem) {
        System.out.println(getClass().getSimpleName() + " enviando mensagem: " + mensagem);
        mediador.enviarMensagem(mensagem, this);
    }

    public abstract void receberMensagem(String mensagem);
}
```

```java
// Implementação específica dos participantes
class Desenvolvedor extends Participante {
    public void receberMensagem(String mensagem) {
        System.out.println("Desenvolvedor recebeu: " + mensagem);
    }
}

class Gerente extends Participante {
    public void receberMensagem(String mensagem) {
        System.out.println("Gerente recebeu: " + mensagem);
    }
}
```

```java
// Testando o padrão Mediator
public class Main {
    public static void main(String[] args) {
        MediadorConcreto mediador = new MediadorConcreto();

        Participante dev = new Desenvolvedor();
        Participante gerente = new Gerente();

        mediador.registrarParticipante(dev);
        mediador.registrarParticipante(gerente);

        dev.enviarMensagem("O código precisa ser revisado!");
        gerente.enviarMensagem("Reunião marcada para 15h.");
    }
}

```